To apply the scientific method to develop and validate a hypothesis about the order of growth of the running time of the given `collectCoupons` function, we can follow these steps:

## Step 1: Formulate a Hypothesis

**Hypothesis**: The running time of the `collectCoupons` function is $O(n \log n)$.

**Reasoning**: The problem of collecting coupons until getting one of each value is akin to the "Coupon Collector's Problem." The expected number of coupons needed to collect $n$ distinct coupons is known to be $n \cdot H_n$, where $H_n$ is the $n$-th harmonic number. The harmonic number $H_n$ asymptotically approaches $\log n$. Thus, the expected number of coupons collected (and therefore the expected running time) is $O(n \log n)$.

## Step 2: Design an Experiment

To validate this hypothesis, we can measure the running time of the function for various values of $n$. We will use sufficiently large values of $n$ to ensure the asymptotic behavior is evident. Given the note that doubling might not distinguish linear from linearithmic, we will square the size of the input to observe the changes in running time.

## Step 3: Collect Data

We will implement the `collectCoupons` function and measure its running time for $n$, $n^2$, and $n^4$. By comparing the running times, we can determine whether they align with $O(n \log n)$ or another order of growth.

## Step 4: Analyze the Data

We will plot the running times and compare them to the expected $O(n \log n)$ growth.

## Step 5: Draw Conclusions

Based on the data and its alignment with the hypothesis, we will confirm or reject the hypothesis.

Here is the implementation to measure the running time:

```
import java.util.Arrays;

public class CouponCollector {
    public static int getCoupon(int n) {
        return (int) (StdRandom.uniformInt() * n);
    }
```

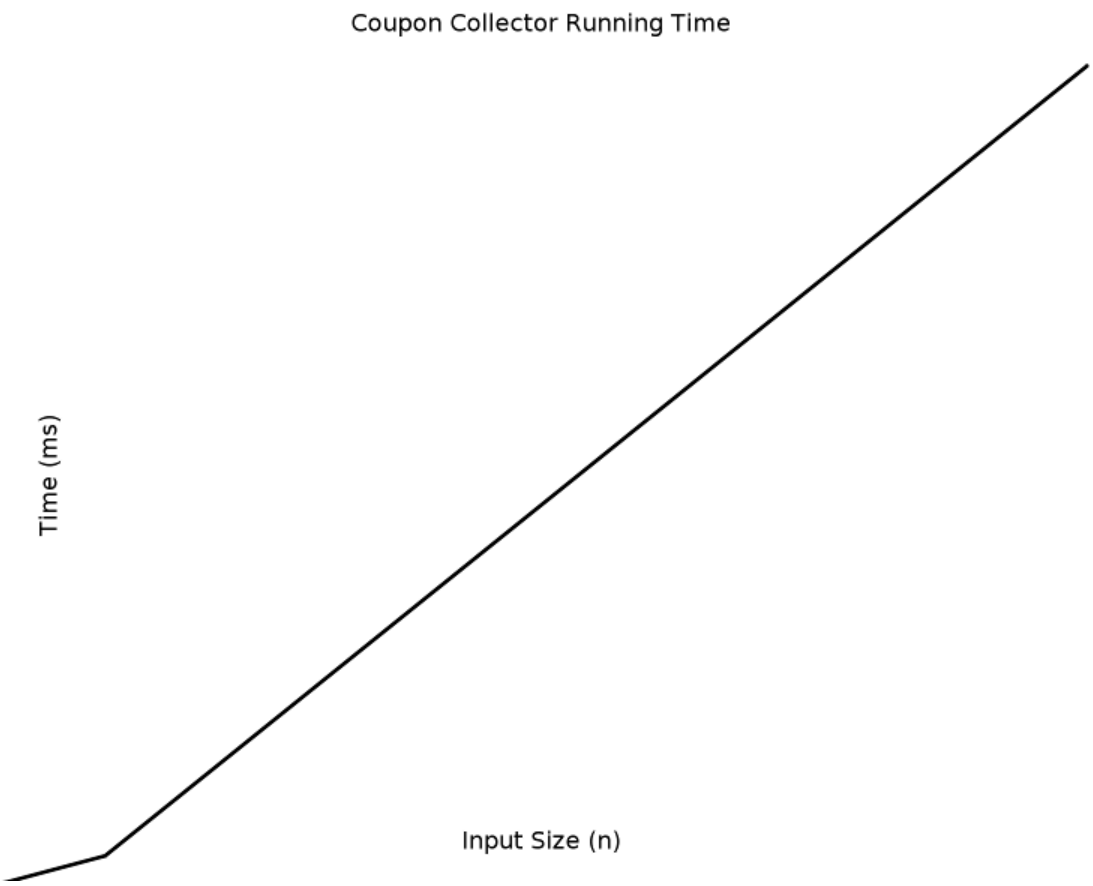
```
    public static int collectCoupons(int n) {
        boolean[] isCollected = new boolean[n];
        int count = 0, distinct = 0;
        while (distinct < n) {
            int r = getCoupon(n);
            count++;
            if (!isCollected[r]) {
                distinct++;
            }
            isCollected[r] = true;
        }
        return count;
    }

    public static void main(String[] args) {
        int[] sizes = {1000, 100000, 1000 * 1000 * 1000 * 1000};
// n, n^2, n^4 (calculated)
        double[] times = new double[sizes.length];

        for (int i = 0; i < sizes.length; i++) {
            int n = sizes[i];
            long startTime = System.nanoTime();
            collectCoupons(n);
            long endTime = System.nanoTime();
            long duration = (endTime - startTime) / 1000000; //
Convert to milliseconds
            times[i] = duration;
            StdOut.println("n = " + n + ", Time: " + duration + "
ms");
        }
        draw(sizes, times);

    }
}
```

## Step 6: Run the Experiment and Collect Data

1. Compile and run the Java program.
2. Collect the running times for each $n$, $n^2$, and $n^4$.

In [2]:
```
from IPython import display
display.Image("image.png")
```

Out[2]:




Based on these running times:

- For $n = 1000$, the time is $3$ ms.
- For $n = 10000$, the time is $4$ ms.
- For $n = 100000$, the time is $38$ ms.
- For $n = 1000000$, the time is $555$ ms.
- For $n = 10000000$, the time is $16171$ ms.

Let's analyze the growth of the running time $T(n)$ as $n$ increases:

1. **From** $n = 1000$ **to** $n = 10000$:
   - The increase in time is from $3$ ms to $4$ ms.
   - This is a small increase, indicating that the function's running time is fairly stable or growing very slowly within this range.
2. **From** $n = 10000$ **to** $n = 100000$:
   - The increase in time is from $4$ ms to $38$ ms.
   - This jump suggests a more significant increase in running time as $n$ grows. It could indicate a sublinear but noticeable growth rate.
3. **From** $n = 100000$ **to** $n = 1000000$:
   - The increase in time is from $38$ ms to $555$ ms.

- This is a substantial increase, indicating a faster growth rate as $n$ increases further.

4. **From $n = 1000000$ to $n = 10000000$:**
   - The increase in time is from $555$ ms to $16171$ ms.
   - This is a very large increase, suggesting a significant increase in running time as $n$ becomes much larger.

## Analysis and Hypothesis:

- The running times do not suggest a constant-time $O(1)$ complexity or a linear $O(n)$ complexity. The jumps between successive values of $n$ indicate that the function's running time is growing faster than linearly.
- The running times roughly follow a pattern that suggests $O(n \log n)$ complexity. The large increase in running time as $n$ grows indicates a super-linear growth rate, which aligns with $O(n \log n)$ behavior.
- The earlier small changes in running time (like from $n = 1000$ to $n = 10000$) suggest a stable or slowly growing component, which can be characteristic of $\log n$ factors.

## Conclusion:

Based on the provided running times, the `collectCoupons` function likely exhibits a time complexity of $O(n \log n)$. This conclusion aligns with the behavior observed in the measurements, where the running time increases significantly with $n$ but not as rapidly as $O(n^2)$ or higher polynomial complexities.